

# Latency Bottleneck Analysis by Ftrace

**Kouta Okamoto**

Advanced Software Technology Group  
Corporate Software Engineering Center  
**TOSHIBA CORPORATION**

2011/5/20

# 本日の発表のアウトライン

---

- はじめに
- ボトルネック箇所特定調査事例1
  - 環境
  - 調査内容
  - 調査結果
- ボトルネック箇所特定調査事例2
  - 環境
  - 調査内容
  - 調査結果
- まとめ

# はじめに

---

## ■ Linuxカーネルのレイテンシ

- cyclicttestで周期起動のレイテンシの測定が可能

しかし・・・

- 性能が悪かった場合
  - どこがボトルネックとなっているか検証する必要がある

そこで！！

実際にボトルネック箇所を検証した結果を紹介



# ボトルネック箇所特定調査事例1 ~ARM~

## ■ 測定環境

### ■ CPU

- ARM Cortex-A8(800MHz)

### ■ メモリ

- 512MB

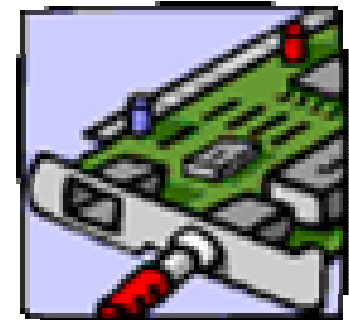
### ■ カーネル

- Linux 2.6.31.12

- CONFIG\_HZ=100
- CONFIG\_SWAP=n
- CONFIG\_PREEMPT=y
- CONFIG\_HIGH\_RES\_TIMER=y
- CONFIG\_NO\_HZ=n
- 電源管理周りのコンフィグレーションをACPI以外無効化

### ■ ユーザ環境

- Debian GNU/Linux 6.0.0(squeeze)
- `echo -1 > /proc/sys/kernel/sched_rt_runtime_us`



# ボトルネック箇所特定調査事例1 ~ARM~

## ■ 測定プログラム

### ■ cyclicttest

#### ■ 周期

- 300  $\mu$  秒、500  $\mu$  秒、1000  $\mu$  秒

#### ■ 試行回数

- 100万回

#### ■ 実行コマンド

- `$ ./cyclicttest -q -m -i周期 -p99 -l1000000 -h1000`

### ■ 負荷(測定プログラム以外の負荷)

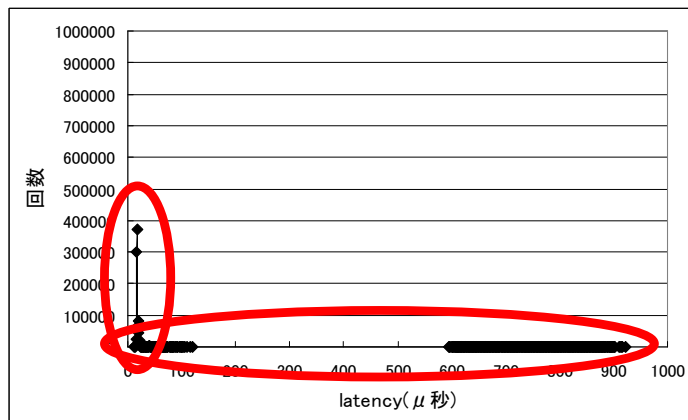
#### ■ CPU負荷無し

#### ■ CPU負荷約60%(TOPで確認)

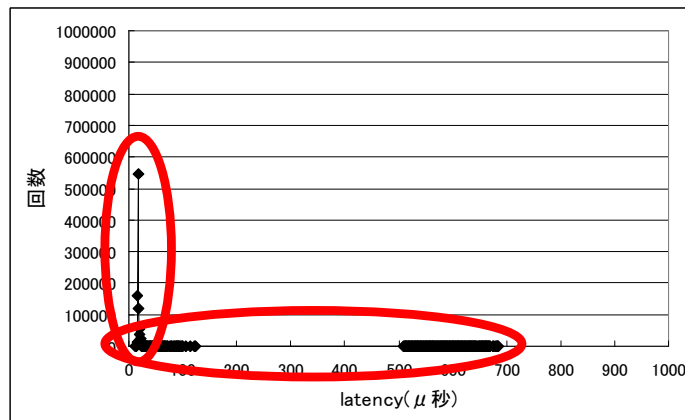


# 測定結果

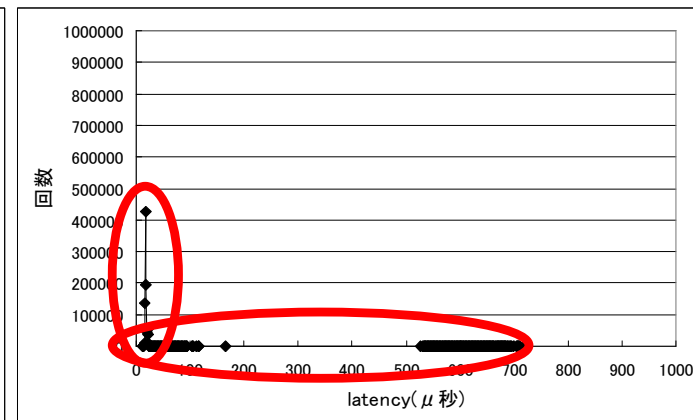
## ■ CPU負荷無し



300 μ 秒周期



500 μ 秒周期

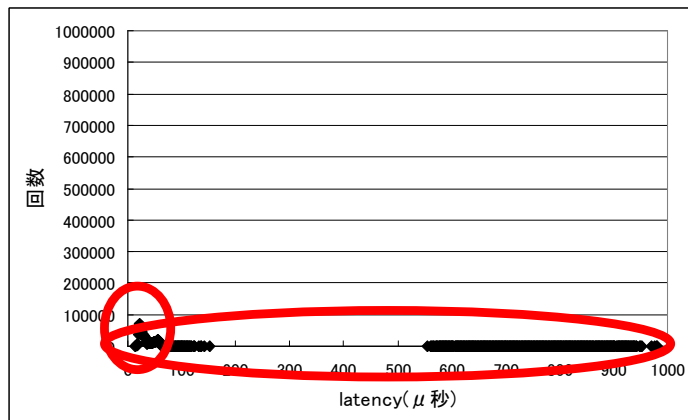


1000 μ 秒周期

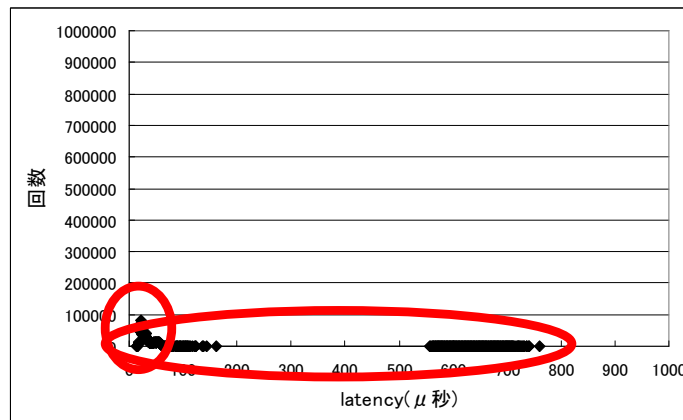
周期	最小レイテンシ	平均レイテンシ	最大レイテンシ	周期オーバー回数	標準偏差
300 μ 秒	11	19.025	921	3018	40.476
500 μ 秒	11	19.458	684	5026	39.713
1000 μ 秒	11	23.011	717	0	57.798

# 測定結果

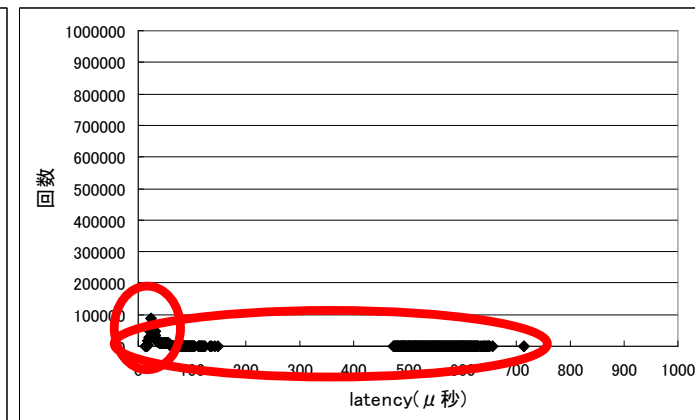
## ■ CPU負荷60%



300 μ 秒周期



500 μ 秒周期



1000 μ 秒周期

周期	最小レイテンシ	平均レイテンシ	最大レイテンシ	周期オーバー回数	標準偏差
300 μ 秒	13	34.165	980	3018	41.386
500 μ 秒	13	33.234	760	5025	44.218
1000 μ 秒	12	35.014	714	0	52.458

負荷の有無に関らず最大レイテンシが1ms近い値



ボトルネックはどこ？

# ボトルネックの特定

---

## ■ Function Graph Tracerの利用

### ■ 機能

- 関数の呼び出しと実行時間をトレース(サブ関数の実行時間含む)
- C言語ライクに関数をインデントして表示してくれる

### ■ パッチ適用

- ARM向けカーネル2.6.31.12では標準では利用不可
- elinux.orgから提供されている2.6.31-rc1向けFunction Graph Tracerパッチをマージ

URL: [http://elinux.org/Ftrace\\_Function\\_Graph\\_ARM](http://elinux.org/Ftrace_Function_Graph_ARM)

## ■ cyclicttestの修正

- cyclicttestにthreshold機能を付加
  - レイテンシがthresholdを超えたら測定を終了



# ボトルネックの特定

---

## ■ トレースの実行

# RTタスクの実行時間制限解除

```
$ echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

# function graph tracerの初期化

```
$ mount -t debugfs nodev /sys/kernel/debug
```

```
$ echo 0 > /sys/kernel/debug/tracing/tracing_enabled
```

```
$ echo function_graph > /sys/kernel/debug/tracing/current_tracer
```

```
$ echo funcgraph-abstime > /sys/kernel/debug/tracing/trace_options
```

```
$ echo 128000 > /sys/kernel/debug/tracing/buffer_size_kb
```

# 実行

```
$ echo 1 > /sys/kernel/debug/tracing/tracing_enabled && ¥
```

```
./cyclictest -q -m -i300 -p99 -H閾値 -l1000000 -h1000 && ¥
```

```
echo 0 > /sys/kernel/debug/tracing/tracing_enabled
```

# トレース結果の取得

```
$ cat /sys/kernel/debug/tracing/trace > trace-`uname -r`.log
```

# ボトルネックの特定

## ■ トレースログの検証

- トレースログの末尾から`sys_clock_gettime()`を逆検索  
⇒ 閾値を超えた時点でcyclictestを終了させているため

504.294647		0)		<code>sys_timer_getoverrun()</code> {
504.294649		0)		<code>lock_timer()</code> ;
504.294654		0)		}
504.294656		0)		<code>sys_rt_sigtimedwait()</code> {
504.294658		0)		<code>dequeue_signal()</code> {
504.294660		0)		<code>__dequeue_signal()</code> {
504.294662		0)		<code>next_signal()</code> ;
		.		
		.		
		.		
504.297163		0)		<code>recalc_sigpending()</code> {
504.297165		0)		<code>recalc_sigpending_tsk()</code> {
504.297168		0)		}
504.297170		0)		}
				<code>sys_clock_gettime()</code> {
504.297175		0)		<code>posix_ktime_get_ts()</code> {
504.297177		0)		<code>ktime_get_ts()</code> {
504.297178		0)		<code>asm_do_IRQ()</code> {
504.297181		0)		

2513.625 us

ftraceによる  
レイテンシ増加を  
加味

# ボトルネックの特定

## ■ トレースログの検証

- `sys_rt_sigtimedwait()`関数の開始～終了までの間で遅延増大の原因を調査

504.295229		0)	
504.295232		0)	
504.295234		0)	1.750 us
504.295239		0)	1.750 us
504.295243		0)	
504.295245		0)	
504.295250		0)	1.875 us
504.295256		0)	4.625 us
504.295265		0)	3.375 us
504.295272		0)	3.250 us
504.295279		0)	3.250 us
504.295286		0)	3.625 us
504.295292		0)	3.375 us
504.295299		0)	3.375 us
504.295305		0)	3.375 us
504.295312		0)	3.250 us
504.295319		0)	3.250 us
504.295325		0)	3.375 us

```
__do_softirq() {  
    run_timer_softirq() {  
        hrtimer_run_pending();  
        preempt_schedule();  
        ehci_watchdog() {  
            ehci_work() {  
                free_cached_itd_list();  
                qh_completions();  
                qh_completions();  
                qh_completions();  
                qh_completions();  
                qh_completions();  
                qh_completions();  
                qh_completions();  
                qh_completions();  
                qh_completions();  
                qh_completions();  
                qh_completions();  
            }  
        }  
    }  
}
```

USBからの  
データ取得  
のために  
遅延が増大

# ボトルネックの特定

## ■ 原因

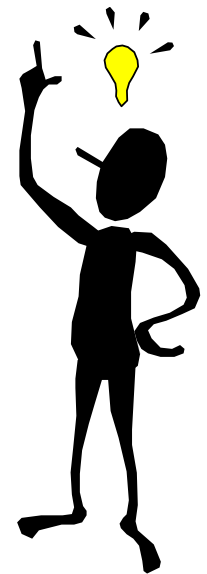
- USBからデータを取得するタイミングで遅延が増大
- USBのデータ取得はポーリング(現状100ms毎)



## ■ 対策

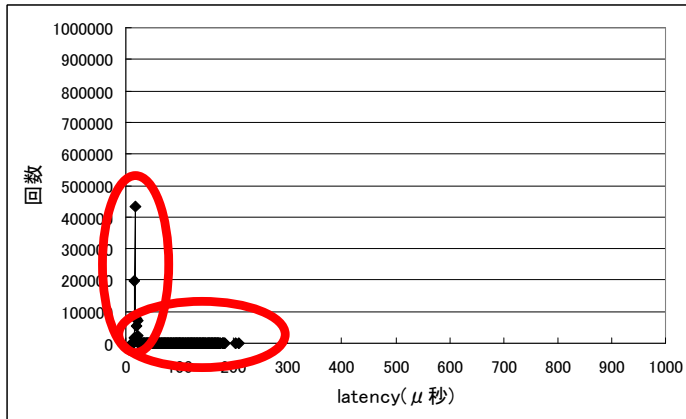
- USBのポーリング間隔を短くする(10ms毎)
  - 1回のデータ取得に対するデータ量を小さくする

```
diff --git a/drivers/usb/host/ehci-hcd.c b/drivers/usb/host/ehci-hcd.c
index 0c9b7d2..db2efd2 100644
--- a/drivers/usb/host/ehci-hcd.c
+++ b/drivers/usb/host/ehci-hcd.c
@@ -83,7 +83,7 @@ static const char hcd_name [] = "ehci_hcd";
#define EHCI_TUNE_FLS 2 /* (small) 256 frame schedule */
#define EHCI_IAA_MSECS 10 /* arbitrary */
-#define EHCI_IO_JIFFIES (HZ/10) /* io watchdog > irq_thresh */
+#define EHCI_IO_JIFFIES (HZ/100) /* io watchdog > irq_thresh */
#define EHCI_ASYNC_JIFFIES (HZ/20) /* async idle timeout */
#define EHCI_SHRINK_FRAMES 5 /* async qh unlink delay */
```

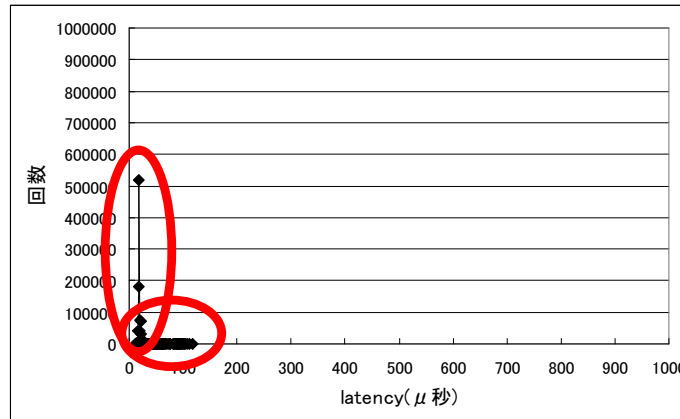


# 測定結果2(1/2)

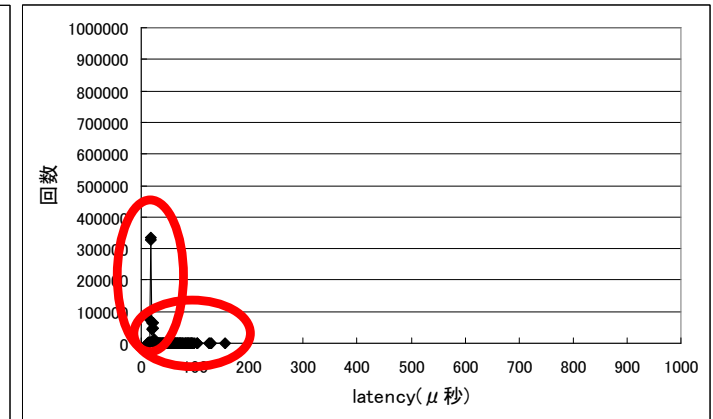
## ■ CPU負荷無し



300 μ 秒周期



500 μ 秒周期

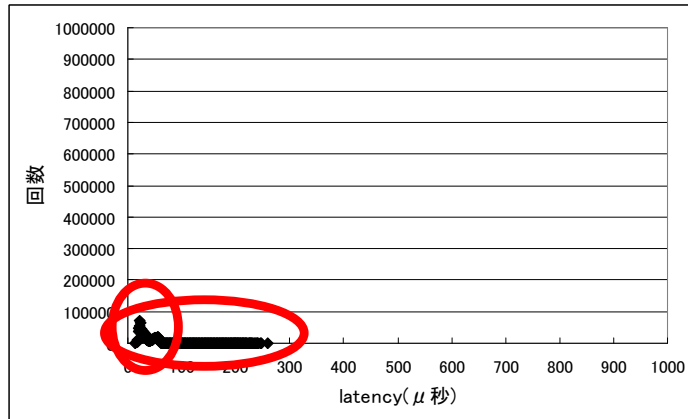


1000 μ 秒周期

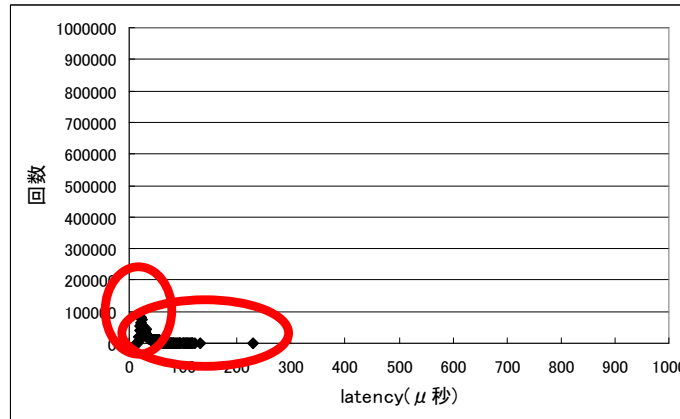
周期	最小レイテンシ	平均レイテンシ	最大レイテンシ	周期オーバー回数	標準偏差
300 μ 秒	11	17.590	209	0	7.925
500 μ 秒	11	17.583	117	0	2.921
1000 μ 秒	11	17.610	154	0	4.081

# 測定結果2(2/2)

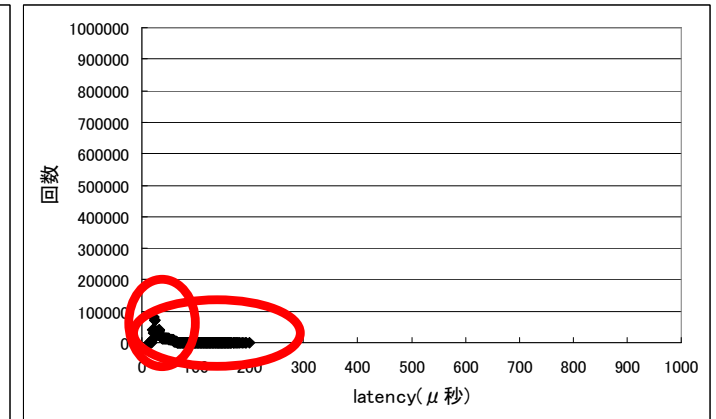
## ■ CPU負荷60%



300 μ 秒周期



500 μ 秒周期



1000 μ 秒周期

周期	最小レイテンシ	平均レイテンシ	最大レイテンシ	周期オーバー回数	標準偏差
300 μ 秒	13	33.365	259	0	16.771
500 μ 秒	13	29.695	228	0	10.782
1000 μ 秒	12	33.222	199	0	14.501

最大レイテンシを大幅に抑えることに成功！！

# 調査結果

---

## ■ ボトルネック箇所検証

- 以下のツールや機能を利用してボトルネック箇所を特定
  - **Function Graph Tracer**
    - ARM向けカーネル2.6.31.12へパッチをマージ
  - **cyclicttest**
    - threshold機能を付加
- USBのポーリングでボトルネックが発生していることを確認
  - ポーリング間隔を短くすることで最大レイテンシの低減に成功

# ボトルネック箇所特定調査事例2 ~x86~

---

- Jamboree 33, 34で各種リアルタイム拡張を用いた周期タスクのレイテンシ評価の話をしました
  - [http://elinux.org/images/3/33/Verification\\_of\\_response\\_time-20100604.pdf](http://elinux.org/images/3/33/Verification_of_response_time-20100604.pdf)
  - <http://elinux.org/images/9/9d/EvaluationOfResponseTimeWithMemoryAccessLoad-Yoshi.pdf>
  
- こんな話がありました・・・
  - 30msの遅延が生じるケースがある
    - 原因については現在未確認
    - 根本的に違う場所で発生している可能性が高い
  
- ということで、やってみました。
  - ARMでのボトルネック調査と同じ手法



# ボトルネック箇所特定調査事例2 ~x86~

## ■ 前回の測定環境

### ■ CPU

- Intel Pentium 4 (2.66GHz)

### ■ メモリ

- 512MB

### ■ カーネル

- Linux 2.6.31.12-rt21

- CONFIG\_HZ=1000

- CONFIG\_SWAP=n

- CONFIG\_PREEMPT\_RT=y

- CONFIG\_HIGH\_RES\_TIMER=y

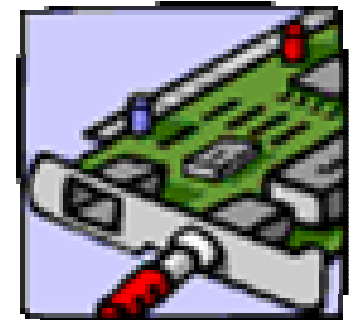
- CONFIG\_NO\_HZ=n

- 電源管理周りのコンフィグレーションをACPI以外無効化

### ■ ユーザ環境

- Debian GNU/Linux 5.0(lenny)

- `echo -1 > /proc/sys/kernel/sched_rt_runtime_us`



# ボトルネック箇所特定調査事例2 ~x86~

## ■ 前回の測定プログラムのおさらい

### ■ 自家製周期実行プログラム

#### ■ 周期

■ 300  $\mu$  秒、500  $\mu$  秒、1000  $\mu$  秒

#### ■ 試行回数

■ 10万回

### ■ 負荷(測定プログラム以外の負荷)

■ CPU負荷無し

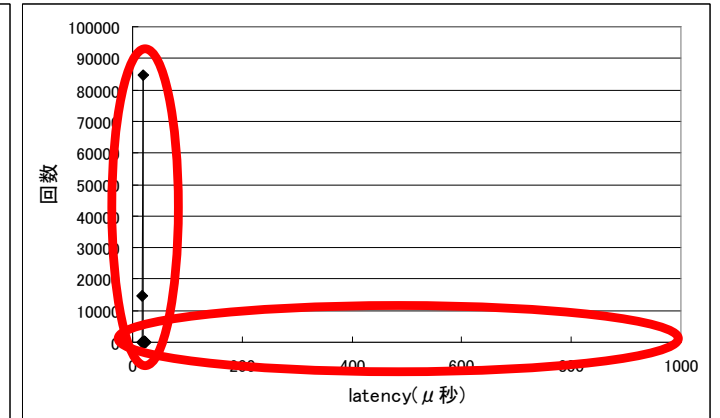
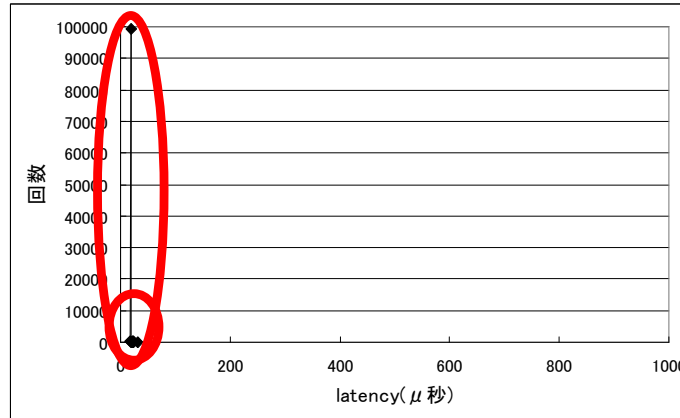
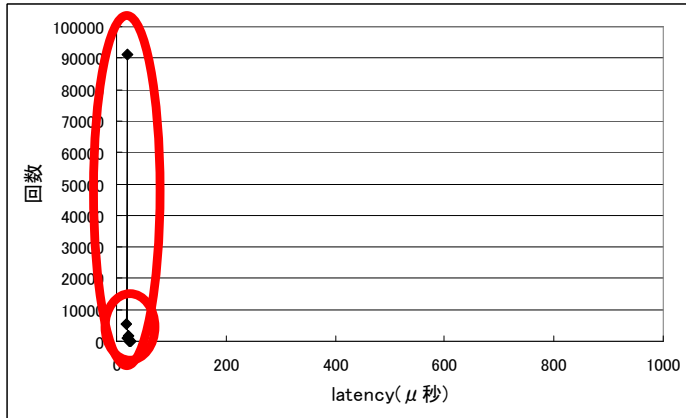
■ CPU負荷約50%(TOPで確認)

■ CPU負荷約100%(TOPで確認)



# 前回の測定結果

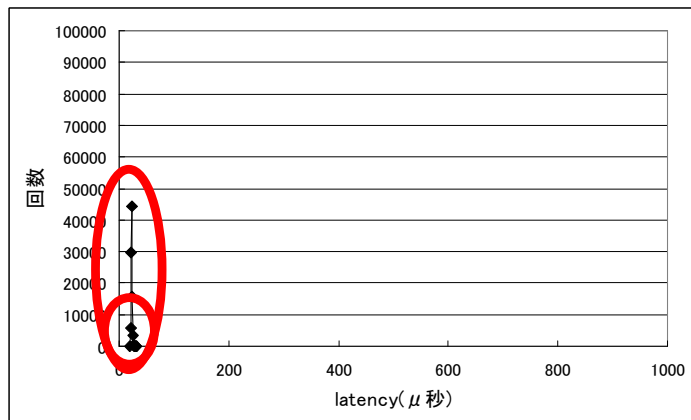
## ■ CPU負荷無し



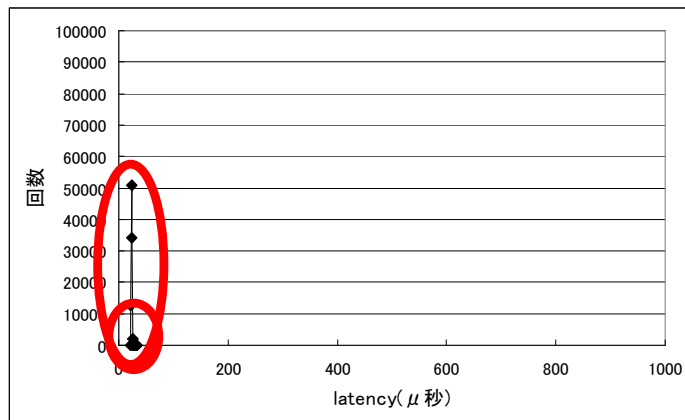
周期	最小レイテンシ	平均レイテンシ	最大レイテンシ	周期オーバー回数	標準偏差
300 $\mu$ 秒	18.118	19.162	25.629	0	0.422
500 $\mu$ 秒	18.171	19.269	32.615	0	0.246
1000 $\mu$ 秒	17.935	19.361	27207.563	1	12.593

# 前回の測定結果

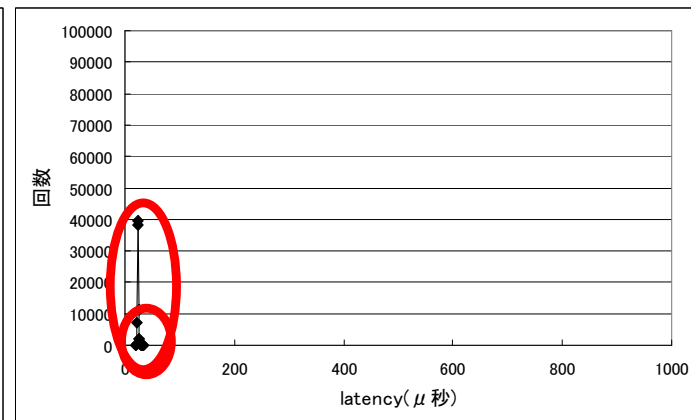
## ■ CPU負荷50%



300  $\mu$  秒周期



500  $\mu$  秒周期

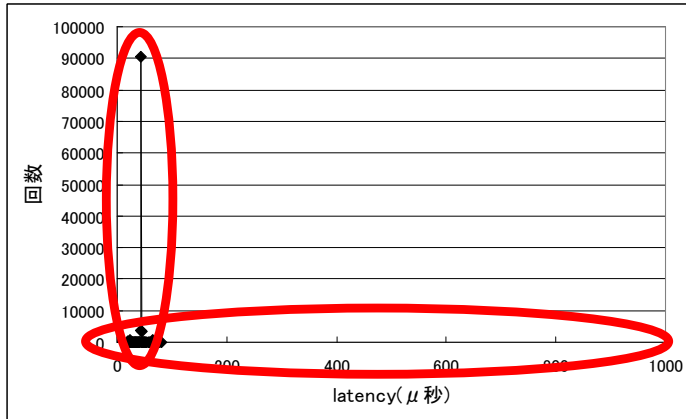


1000  $\mu$  秒周期

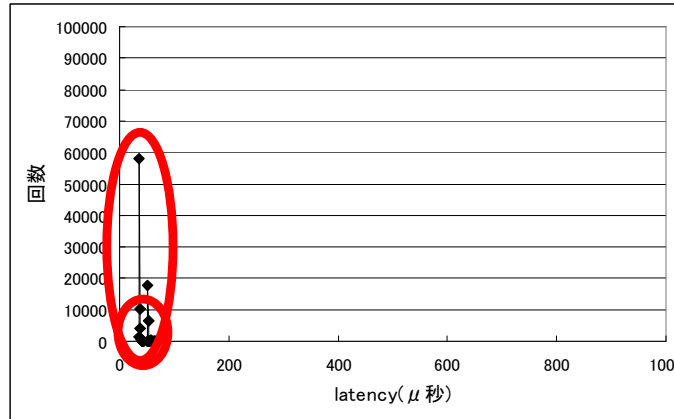
周期	最小レイテンシ	平均レイテンシ	最大レイテンシ	周期オーバー回数	標準偏差
300 $\mu$ 秒	20.488	23.340	32.353	0	0.979
500 $\mu$ 秒	21.229	23.763	33.263	0	0.802
1000 $\mu$ 秒	20.616	24.162	34.459	0	1.041

# 前回の測定結果

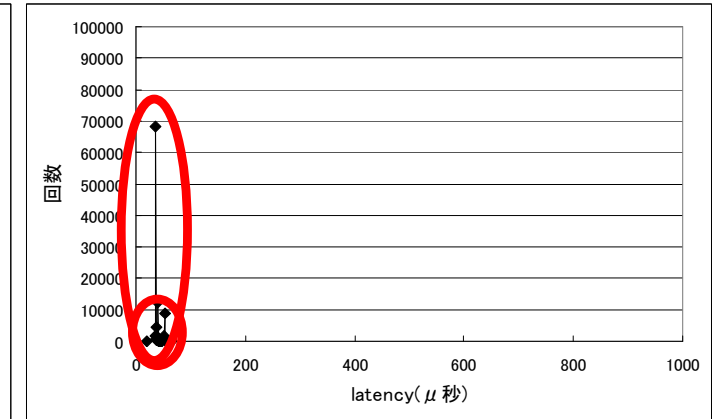
## ■ CPU負荷100%



300  $\mu$  秒周期



500  $\mu$  秒周期



1000  $\mu$  秒周期

周期	最小レイテンシ	平均レイテンシ	最大レイテンシ	周期オーバー回数	標準偏差
300 $\mu$ 秒	19.292	44.624	28282.187	1	12.87
500 $\mu$ 秒	35.462	40.802	69.411	0	7.02
1000 $\mu$ 秒	19.376	39.129	70.020	0	5.369

低頻度で30ms近い遅延が発生していた



ftraceで調査！！

# ボトルネックの特定

## ■ Function Graph Tracerを利用したトレース結果

- トレースログの末尾からsys\_clock\_gettime()を逆検索
- sys\_rt\_sigtimedwait()関数の開始～終了までの処理を調査

5586.207717		0)			activate_task() {
5586.207718		0)			enqueue_task() {
5586.207718		0)			enqueue_task_rt() {
5586.207718		0)			cpupri_set() {
5586.207719		0)	0.407 us		_atomic_spin_lock_irqsave();
5586.207720		0)	0.448 us		_atomic_spin_unlock_irqrestore();
5586.207721		0)	0.430 us		_atomic_spin_lock_irqsave();
5586.207722		0)	0.444 us		_atomic_spin_unlock_irqrestore();
5586.207723		0)	4.391 us		}
5586.207723		0)	0.394 us		update_rt_migration();
5586.207724		0)	6.158 us		}
5586.207725		0)	7.040 us		}
5586.235835		0)	! 28117.53 us		}
5586.235836		0)			check_preempt_curr_rt() {
5586.235836		0)	0.415 us		resched_task();
5586.235837		0)	1.397 us		}

ここでレイテンシ発生！！

# ボトルネックの特定

## ■ ソースコードの確認

【kernel/sched.c】

```
static void inc_nr_running(struct rq *rq)
{
    rq->nr_running++;
}

static void
activate_task(struct rq *rq, struct task_struct *p, int
wakeup, bool head)
{
    if (task_contributes_to_load(p))
        rq->nr_uninterruptible--;

    enqueue_task(rq, p, wakeup, head);
    inc_nr_running(rq);
}
```

単なるインクリメント

この箇所で  
レイテンシが発生



大きなレイテンシは発生しそうにない・・・！？

# ボトルネックの特定

## ■ Function Graph Tracerを利用したトレース結果2

### ■ 同じ方法で再度検証

184.976213		0)	0.404 us		pre_schedule_rt();
184.976214		0)			put_prev_task_rt() {
184.976214		0)			update_curr_rt() {
185.004323		0)	0.537 us		sched_avg_update();
185.004324		0)	! 28109.76 us		}
185.004324		0)	! 28110.62 us		}
185.004325		0)	0.496 us		pick_next_task_rt();
185.004326		0)	0.442 us		perf_counter_task_sched_out();
185.004327		0)	0.434 us		native_load_sp0();
185.004328		0)	0.465 us		native_load_tls();

ここでレイテンシ発生！！



# ボトルネックの特定

## ■ ソースコードの確認

【kernel/sched.c】

```
static void update_curr_rt(struct rq *rq)
{
    struct task_struct *curr = rq->curr;
    struct sched_rt_entity *rt_se = &curr->rt;
    struct rt_rq *rt_rq = rt_rq_of_se(rt_se);
    u64 delta_exec;

    if (!task_has_rt_policy(curr))
        return;

    delta_exec = rq->clock - curr->se.exec_start;
    if (unlikely((s64)delta_exec < 0))
        delta_exec = 0;

    schedstat_set(curr->se.exec_max, max(curr->se.exec_max, delta_exec));

    curr->se.sum_exec_runtime += delta_exec;
    account_group_exec_runtime(curr, delta_exec);

    curr->se.exec_start = rq->clock;
    cpuacct_charge(curr, delta_exec);

    sched_rt_avg_update(rq, delta_exec);
}
```

この箇所のどこかで  
レイテンシが発生



先程の検証と箇所が変わっている・・・？！！

# ボトルネックの特定

## ■ Function Graph Tracerを利用したトレース結果3

### ■ 同じ方法で再度検証

641.941738		0)		schedule() {
641.941738		0)		__schedule() {
641.941739		0)	0.421 us	rcu_qsctr_inc();
...				
641.941751		0)		pre_schedule_rt() {
641.941752		0)	0.418 us	pull_rt_task();
641.941752		0)	1.281 us	}
641.941753		0)		put_prev_task_rt() {
641.941753		0)		update_curr_rt() {
641.941754		0)	0.426 us	sched_avg_update();
641.941755		0)	1.310 us	}
641.941755		0)	2.178 us	}
641.941756		0)	0.423 us	pick_next_task_fair();
641.969863		0)	0.839 us	pick_next_task_rt();
641.969864		0)	0.422 us	pick_next_task_fair();
641.969865		0)	0.421 us	pick_next_task_idle();
641.969866		0)	0.461 us	perf_counter_task_sched_out();

ここでレイテンシ発生！！

# ボトルネックの特定

## ■ ソースコードの確認

【kernel/sched.c】

```
static inline struct task_struct *  
pick_next_task(struct rq *rq)  
{  
    . . .  
    if (likely(rq->nr_running == rq->cfs.nr_running)) {  
        p = fair_sched_class.pick_next_task(rq);  
        if (likely(p))  
            return p;  
    }  
  
    class = sched_class_highest;  
    for ( ; ; ) {  
        p = class->pick_next_task(rq);  
        if (p)  
            return p;  
        . . .  
        class = class->next;  
    }  
}  
  
asmlinkage void __sched __schedule(void)  
{  
    . . .  
    put_prev_task(rq, prev);  
    next = pick_next_task(rq);
```

この箇所のでどこかで  
レイテンシが発生

また箇所が  
変わっている！！

# ボトルネックの特定

---

## ■ レイテンシ発生箇所

- 測定するごとに箇所が変わる
  - ボトルネック箇所の特定不可
  - 基本的にはスケジューラ周り

## ■ 原因

- 結局のところ不明
- ハードの制約(または不具合)の可能性も考えられる
  - むしろこの可能性のほうが高い？

ちなみに・・・

他のハードで試した場合発生しない

- **Linuxカーネルのレイテンシのボトルネック解析**
  - 以下のツールを利用した解析が可能
    - Function Graph Tracer
      - カーネルバージョンによってはパッチをマージする必要あり
    - cyclicttest
      - 解析のために閾値設定できるよう修正
  - ARMでのボトルネック解析
    - USBのポーリングでボトルネック発生
    - ポーリング間隔を短くすることでボトルネック解消
  - x86でのボトルネック解析
    - なぜかボトルネック箇所が検証の度に変わる
    - ハード依存(不具合?)の可能性大
      - 他のハードでは発生しない

**TOSHIBA**

**Leading Innovation >>>**